# All the rest

## Reading the relations from CSV (relations.py)

```
from collections import namedtuple

#Function to read a file representing a relation
#It takes two attributes
#   fname = The path to the file to read
#   fields = The fields of the relation
#It returns a triple representing a relation of the form:
#   (FIELDS, TUPLES IN RELATION, FUNCTION)
#
#   FUNCTION is needed to decorate a tuple with relation's attributes
def relation_read_file(fname, fields):
    #A relation is a simple hash-table thus we get O(1) insertion, O(1) deletion and we can
    #also traverse it
    relation = set()
    #Builds the decorating function
    record = namedtuple('Record', fields)
    with open(fname) as f:
        for line in f:
            row = record(*[int(field) for field in line.strip().split(",")])
            relation.add(row)
    #Returns the relation
    return (fields, relation, record)

#This is the default database instance, it reads R1, R2, R3 and R4
#in the Data directory
def default_db_instance():
    #Database is represented through a hashtable of the form
    #   Relation Name -> Relation Data
    db = {}
    #The relation files have this structure Rx.txt with x being a number
    fname = lambda x : "./Data/R" + x + ".txt"
    db['1'] = relation_read_file(fname("1"), ("A", "B"))
    db['2'] = relation_read_file(fname("2"), ("A", "C"))
    db['3'] = relation_read_file(fname("3"), ("B", "C"))
    db['4'] = relation_read_file(fname("4"), ("A", "D"))
    return db
```

# Join query implementation (join.py)

```
from collections import *
from relations import *

FIELDS, RECORDS = 0, 1

#Implementation of a binary hash-join
#designed for IncDB
def h_binary_join(r1, r2):
    if len(r1[RECORDS]) > len(r2[RECORDS]):
        #The outer relation must the smallest one
        #so we create the hash-table of joining attributes
        #for the smallest relation
        tmp = r1
        r1 = r2
        r2 = tmp
    #The relation records
    r1_rel, r2_rel = r1[RECORDS], r2[RECORDS]
    #Descriptors containing the relations attribute names
    r1_fields = set(r1[FIELDS])
    r2_fields = set(r2[FIELDS])
    #Obtain the joining attributes of the to relations to be joined
    joining_attributes = r1_fields.intersection(r2_fields)
    #List of the attributes in the result
    result_attributes = tuple(r1_fields.union(r2_fields))
    #A functon to add named attributes to tuple records so we can access
    #them by their attribute's name
    record = namedtuple('Record', result_attributes)
    #The relation holding the result of the query
    result_relation = set()
    #Buckets creation of the outer relation
    buckets = defaultdict(list)
    for rec in r1_rel:
        j_attr = tuple(rec.__dict__[attr] for attr in joining_attributes)
        buckets[j_attr].append(rec)
    #Iterate through the every record in the smaller relation
    for r2_rec in r2_rel:
        #Obtain the values of the joining attributes
        j_attr = tuple(r2_rec.__dict__[attr] for attr in joining_attributes)
        #Check if there is match
        if j_attr in buckets:
            #There is matching bucket for the joining attribute of the inner relation
            #Iterates every element falling in the bucket and at every iteration
            #creates a new result record
            for r1_rec in buckets[j_attr]:
                #Make new result record
                fields = [getattr(r1_rec, attr) if attr in r1_rec.__dict__ else
getattr(r2_rec, attr) for attr in result_attributes]
                #Adds that to the result relation
                result_relation.add(record(*fields))
    #Returns the relation holding the result
    return (result_attributes, result_relation, record)

#Implementation of a binary hash-join
#designed for NaiveDB
def sm_binary_join(r1, r2):
    if len(r1[RECORDS]) > len(r2[RECORDS]):
        #The outer relation must the smallest one
        #so we create the hash-table of joining attributes
        #for the smallest relation
        tmp = r1
```

```
        r1 = r2
        r2 = tmp
    #The records inside both the relations
    r1_rel, r2_rel = r1[RECORDS], r2[RECORDS]
    #Descriptor containing the field names
    r1_fields = set(r1[FIELDS])
    r2_fields = set(r2[FIELDS])
    #The attribute on which the join is going to performed
    joining_attributes = r1_fields.intersection(r2_fields)
    #Result
    result_attributes = tuple(r1_fields.union(r2_fields))
    record = namedtuple('Record', result_attributes)
    result_relation = set()
    #Function to obtain the joining attribute value of a record
    j_attr_value = lambda x: [getattr(x, attr) for attr in joining_attributes]
    #Sort_Phase
    r1_sorted = sorted(r1[RECORDS], key=j_attr_value)
    r2_sorted = sorted(r2[RECORDS], key=j_attr_value)
    #Merge-phase
    r1_idx = 0
    r2_idx = 0
    #Until we have iterated every record of one of the two relations
    while r1_idx < len(r1_sorted) and r2_idx < len(r2_sorted):
        #The records of the two relations
        r1_rec = r1_sorted[r1_idx]
        r2_rec = r2_sorted[r2_idx]
        #The joining attribute of the two records
        r1_j = j_attr_value(r1_rec)
        r2_j = j_attr_value(r2_rec)
        #We do not have a match
        #so we increase the row number of the relation
        #having the smallest value for the joining attribute
        if r1_j > r2_j:
            r2_idx += 1
        elif r1_j < r2_j:
            r1_idx += 1
        #We have a match
        else:
            #The record to be added in the result
            fields = [getattr(r1_rec, attr) if attr in r1_rec.__dict__ else getattr(r2_rec,
attr) for attr in result_attributes]
            #Update the result
            result_relation.add(record(*fields))
            #Auxiliary counter to the second relation
            #There could be another match through the second relation so
            #we continue iterating and check if there is any
            i = r2_idx
            #Until we iterate every record...
            while i < len(r2_sorted):
                r2_rec = r2_sorted[i]
                r2_j = j_attr_value(r2_rec)
                #We have a match
                if r1_j == r2_j:
                    #Add the record to the result and increments the counter of the second
relation
                    #It handles automatically duplicates as result is represented as a set
                    fields = [getattr(r1_rec, attr) if attr in r1_rec.__dict__ else
getattr(r2_rec, attr) for attr in result_attributes]
                    result_relation.add(record(*fields))
                    i += 1
                else:
                    break
            #Increase the counter of the first relation
            r1_idx += 1
```

```
    #Return the result
    return (result_attributes, result_relation, record)

#Function to execute multi-join
#The parameters are:
#   - The relations to be joined
#   - The join method to use (True = hash-join, False = sort-mergejoin)
def multiway_join(*relations, sort_merge=False):
    #Sort the relations from the smallest one to the biggest one
    sorted_relations = iter(sorted(relations, key= lambda rel: len(rel[RECORDS])))
    #Current semi-join result accumulator
    semi_join = next(sorted_relations)
    join_method = sm_binary_join if sort_merge else h_binary_join
    #Execute a series of semi-joins
    for rel in sorted_relations:
        semi_join = join_method(semi_join, rel)
    return semi_join
```

# IncDB General Class – Extended by representations (IncDB.py)

```
from collections import Counter

#This class is a backbone representing the IncDB's Materialized View engine
class IncDB:

    def __init__(self, database, jrels):
        #Database instance
        self.db = database
        #The joining relations for the supplied database instances
        #Defines the conjunctive query to be materialized
        self.jrels = jrels
        #Obtain the joining attributes of the conjunctive query
        self.jattrs = self.findAttributes([database[rel][0] for rel in jrels])
        #All the attributes of the conjunctive query
        self.attrs = set(attr for rel in jrels for attr in database[rel][0])
        #Non-joining attributes of the non conjunctive query
        self.otherAttrs = tuple(self.attrs.difference(self.jattrs))
        #Some type-casting
        self.attrs = tuple(self.attrs)
        self.jattrs = tuple(self.jattrs)

    #Given a list of attributes of the relations returns the joining attributes
    def findAttributes(self, rel_attributes):
        #Initialize the counter
        c = Counter()
        #For every relation...
        for rel in rel_attributes:
            #...obtain the first attribute and increment it by one in the counter
            c.update({rel[0] : 1})
            #...obtain the second attribute and increment it by one in the counter
            c.update({rel[1] : 1})
        #Return all the attributes that occur more than once
        return [attr for attr in c if c[attr] > 1]

    #This function must be overriden from classes inheriting from IncDB
    #It updates the materialized view of the conjunctive query upon insertion of a record
    def onInsert(self, relation, tuple):
        pass

    #This function must be overriden from classes inheriting from IncDB
    #It updates the materialized view of the conjunctive query upon deletion of a record
    def onDelete(self, relation, tuple):
        pass

    #Updates the materialized view with a list of tuples
    def fill(self, tuples):
        pass

    #Count query
    #Counts the number of rows in the materialized view
    def count(self):
        pass

    #Checks whether a record exists in the current instance of a database
    def existingRecord(self, relation, record):
        return record in self.db[relation][1]

    #Checks whether a record does not exists in the current instance of a database
    def nonExistingRecord(self, relation, record):
        return self.existingRecord(relation, record) == False
```

# NaiveDB Experiment script (Experiment_NaiveDB.py)

```
from FactorizedRepresentation_BinaryJoin import *
from relations import *
from join import *

import copy, time, sys

iname = lambda x: "./Data/I" + str(x) + ".txt"
dname = lambda x: "./Data/D" + str(x) + ".txt"

db_instance = default_db_instance()

test_count = "-count" in sys.argv

conj_query = sys.argv[1::]
conj_query = [rel for rel in conj_query if rel != "-count"]

if len(conj_query) == 0:
    raise Exception("No conjunctive query specified")

time_limit = 60 * 60 # Set the total number of seconds
time_limit_reached = False

print("NAIVE-DB")
print("CONJUNCTIVE QUERY", *["R"+ rel for rel in conj_query])
print("WITH COUNT" if test_count else "WITH NO COUNT")

#Test Insertions
for i in range(1, 6):
    print("TESTING ON INSERTIONS FILE", i)
    db = copy.deepcopy(db_instance)
    elapsedTime = 0
    for line in open(iname(i)):
        els = line.split(",")
        relation = els[0]
        record = (int(els[1]), int(els[2]))
        record = db[relation][2](*record)
        db[relation][1].add(record)
        start = time.time()
        result = multiway_join(*[db[rel] for rel in conj_query], sort_merge = True)
        elapsedTime += (time.time() - start)
        if test_count:
            start = time.time()
            rows = 0
            for row in result[1]:
                rows += 1
            elapsedTime += (time.time() - start)
        if elapsedTime > time_limit:
            time_limit_reached = True
            break

    if time_limit_reached:
        print("Could not process the whole file - ", "Time limit of", time_limit, "reached")
        break
    else:
        print("Processing ", i, "file took", elapsedTime)
    rows = 0
    result = multiway_join(*[db[rel] for rel in conj_query], sort_merge = True)
    for row in result[1]:
        rows += 1
    print("The query result has a total number of", rows, "rows")
```

```
time_limit_reached = False

for i in range(1, 6):
    print("TESTING ON DELETIONS FILE", i)
    db = copy.deepcopy(db_instance)
    elapsedTime = 0
    for line in open(dname(i)):
        els = line.split(",")
        relation = els[0]
        record = (int(els[1]), int(els[2]))
        record = db[relation][2](*record)
        db[relation][1].discard(record)
        start = time.time()
        result = multiway_join(*[db[rel] for rel in conj_query], sort_merge = True)
        elapsedTime += (time.time() - start)
        if test_count:
            start = time.time()
            rows = 0
            for row in result[1]:
                rows += 1
            elapsedTime += (time.time() - start)
        if elapsedTime > time_limit:
            time_limit_reached = True
            break

    if time_limit_reached:
        print("Could not process the whole file - ", "Time limit of", time_limit, "reached")
        break
    else:
        print("Processing ", i, "file took", elapsedTime)
    rows = 0
    result = multiway_join(*[db[rel] for rel in conj_query], sort_merge = True)
    for row in result[1]:
        rows += 1
    print("The query result has a total number of", rows, "rows")
```

# IncDB Experiment script (Experiment_IncDB.py)

```
from FactorizedRepresentation_BinaryJoin import *
from relations import *
from join import *
from ArbitraryQueries_Evaluate import *

import copy, time, sys

iname = lambda x: "./Data/I" + str(x) + ".txt"
dname = lambda x: "./Data/D" + str(x) + ".txt"

db_instance = default_db_instance()

conj_query = sys.argv[1::]
conj_query = [rel for rel in conj_query if rel != "-count"]

test_count = "-count" in sys.argv

if len(conj_query) == 0:
    raise Exception("No conjunctive query specified")

time_limit = 60 * 60 # Set the total number of seconds
time_limit_reached = False

print("INC-DB")
print("CONJUNCTIVE QUERY", *["R"+ rel for rel in conj_query])
print("WITH COUNT" if test_count else "WITH NO COUNT")

IncDB_Engine = FactorizedRepresentation_BinaryJoin if len(conj_query) == 2 else
obtainRepresentation([db_instance[rel] for rel in conj_query])

#Test Insertions
for i in range(1, 6):
    print("TESTING ON INSERTIONS FILE", i)
    db = copy.deepcopy(db_instance)
    initial_result = multiway_join(*[db[rel] for rel in conj_query])
    fdb = IncDB_Engine(db, conj_query)
    fdb.fill(initial_result[1])
    rows = 1
    elapsedTime = 0
    for line in open(iname(i)):
        els = line.split(",")
        relation = els[0]
        record = (int(els[1]), int(els[2]))
        record = db[relation][2](*record)
        start = time.time()
        fdb.onInsert(relation, record)
        elapsedTime += (time.time() - start)
        if test_count:
            start = time.time()
            fdb.count()
            elapsedTime += (time.time() - start)
        db[relation][1].add(record)
        rows += 1
        if elapsedTime > time_limit:
            time_limit_reached = True
            break

    if time_limit_reached:
        print("Could not process the whole file - ", "Time limit of", time_limit, "reached")
        break
```

```
    else:
        print("Processing ", i, "file took", elapsedTime)
    print("The query result has a total number of", fdb.count(), "rows")

time_limit_reached = False

for i in range(1, 6):
    print("TESTING ON DELETIONS FILE", i)
    db = copy.deepcopy(db_instance)
    initial_result = multiway_join(*[db[rel] for rel in conj_query])
    fdb = IncDB_Engine(db, conj_query)
    fdb.fill(initial_result[1])
    rows = 1
    elapsedTime = 0
    for line in open(dname(i)):
        els = line.split(",")
        relation = els[0]
        record = (int(els[1]), int(els[2]))
        record = db[relation][2](*record)
        start = time.time()
        fdb.onDelete(relation, record)
        elapsedTime += (time.time() - start)
        if test_count:
            start = time.time()
            fdb.count()
            elapsedTime += (time.time() - start)
        db[relation][1].discard(record)
        rows += 1
        if elapsedTime > time_limit:
            time_limit_reached = True
            break

    if time_limit_reached:
        print("Could not process the whole file - ", "Time limit of", time_limit, "reached")
        break
    else:
        print("Processing ", i, "file took", elapsedTime)
    print("The query result has a total number of", fdb.count(), "rows")
```